Building a Relocatable Python

Marc Paterno
Revision 1

Contents

1 Introduction

The SSI group distributes most of the software we produce using "relocatable UPS". UPS provides a facility to allow run-time selection between multiple versions of installed "packages", and to prevent the setup of mutually-inconsistent packages. In addition, "relocatable UPS" provides the ability for users with no elevated system privileges to install pre-built binary packages, and simplifies the management of multiple installed packages. Python is one of the many packages delivered through UPS.

The "traditional" UPS manner or building Python (the Python interpreter and its Standard Library) results in a package that is not fully relocatable. While many features of the distributed package work correctly, the user often will experience problems in using *distutils*[1], the Python Standard Library component that provides a build system for Python extension modules.

The problem is that *distutils* has compiled-in (or, as it turns out, code-generated-at-build-time) absolute paths, resulting in the inability for *setup.py* scripts for third-party code (*e.g. numpy*) to find necessary Python libraries.

In this note I describe what needed to be done to fix this problem, so that out-of-the-box builds of third-party code would work. An important goal of this effort was to retain the UPS philosophy of delivering already-built software, rather than software that needs to be built on each platform. This was important for Python since we distribute a pre-built Root package which depends upon a specific Python distribution. If we were unable to deliver a pre-built Python distribution, we would not have an easy and safe means of delivering a Root binary installation package that would depend on a specific Python build.

[1] *distutils* is described at http://docs.python.org/2/library/distutils.html.

Terminology

Module: A Python module is a body of code which may define classes, functions, and constants, and which is loaded using an `import` statement. Modules may be "pure" (written in Python) or "impure" (written in a compiled language, *e.g.* C).

2 How to repeat the problem

To demonstrate the failure of the traditional UPS build of Python, one can follow these steps:

1. Build a Python "source" distribution using the python-ssi-build scripts.
2. Build a Python binary distribution, from the generated "source" distribution. This may be done on the same machine that was used for the creation of the "source" distribution tarball.
3. Install the binary distribution kit on a different machine (not on the machine on which the binary distribution kit was built).
4. Use the installed Python to build an impure extension module. This will fail due to an inability to find `libpython2.7.so`, because the link-line flags used by *distutils* reflect the location at which Python was built, rather than where is was installed.

Figure 1 shows the C-language source code for a very simple impure module. This module, named `helloworld`, contains a single function, called `hello`; the implementation is found in the file `hi.c`.[2] The instructions used to build this module are shown in figure 2. These instructions make use of *distutils*.

To build the extension module, move to a directory containing the two files `hi.c` and `setup.py`, and run:

```
> python setup.py install --user
```

This is intended to build the Python module *helloworld*, and install it in a user-specific "site-packages" directory. The module would then be available through use of the Python `import` statement. However, with an imperfectly-relocatable Python build, the linking of the *helloworld* module will fail. This is the problem for which this document describes the solution.

3 Building a fully relocatable Python

The key observation in the failure is that *distutils* makes use of a (pure) Python module *sysconfigdata*, which in turn makes use of a (pure) Python module *_sysconfigdata*. This can be found in during the build of Python at, *e.g.*, `/v2_7_5a/Linux64bit+2.6-2.12/src/Python-2.7.5/build/lib.linux-x86_64-2.7/_sysconfigdata.py` (for a build

[2] I have chosen this seemingly odd selection of names to make it clear which name is used for which purpose in creating the module. More common naming practice would be for the module, the single function, and the C source code filename to all be the same.

```
#include "Python.h"

static PyObject* helloworld(PyObject* self) {
    return Py_BuildValue("s", "Hello, Python extensions!!");
}

static char hw_docs[] = "helloworld( ): put docs here\n";

static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld, METH_NOARGS, hw_docs},
    {NULL}
};

void inithelloworld(void) {
    Py_InitModule3("helloworld", helloworld_funcs, "Example!");
}
```

Figure 1: The C source code for a simple impure Python module, file `hi.c`.

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0',  \
      ext_modules=[Extension('helloworld', ['hi.c'])])
```

Figure 2: Build instructions for the *helloworld* module, file `setup.py`.

of Python v2.7.5a, on a 64-bit SLF6 machine; on other machines, the location will vary). The source text for this module is written as part of the build of Python. It is the result of parsing several other files (`Makefile`s and configuration files).

This module contains a single definition, for a variable named `build_time_vars`. This variable is assigned a value that is a Python `dict`, with about 500 entries. The keys are all strings, as are the values. Many of the values denote paths on the system on which Python was built.

The central feature of the fix involves modifying the build to patch this file immediately after its generation. The patch rewrites the file so that importing the module sets several local variables to values determined by the values of several environment variables at the time of the import, and to replace the use of hard-coded paths by these variables. A truncated version of the code is shown in figure 3.

The general pattern of the solution is:

1. Insert code to capture the necessary environment variable values at import time.
2. Replace portions of hard-coded paths with these captured values.

The specific environment variables that had to be used are:

- `PYTHON_ROOT`, which points to the "root" of the Python installation; this is the directory under which are found the `bin` and `lib` directories containing the installed

Python product. For example, the Python interpreter is found at `${PYTHON_ROOT}/bin/python`.

- `PYTHON_DIR`, which points to the base of the UPS installation of Python for the particular release; it is the directory under which the UPS flavor/qualifier directories appear, and under which the product's `ups` directory would be found. For example, `${PYTHON_DIR}/ups/python.table` is the location for the table file for that version of Python.
- `SQLITE_DIR` points to the base of the UPS installation for SQLite, similar to the meaning of `PYTHON_DIR`. `SQLITE_DIR` is needed so that we make sure the Python Standard Library *sqlite3* module is built with the same SQLite version as is the other software we develop.

```
1 from os import environ as env
2 this_python_root=env[``PYTHON_ROOT'']
3 this_python_dir=env[``PYTHON_DIR'']
4 this_sqlite_dir=env[``SQLITE_DIR'']
5 build_time_vars = {'AC_APPLE_UNIVERSAL_BUILD': 0,
6 ...
7  'BINDIR': this_python_root+'/bin',
8 ...
```

Figure 3: A portion of the patched `_sysconfigdata.py`, showing the introduction of the import-time evaluation of the environment variables.

4 Conclusions

With the modifications described above, there are no features of Python known to fail. I have tested both building of impure modules and use of the Python debugger.

5 Recommendations

The Python Standard Library includes many modules that require underlying C libraries. We have already had to modify the Python build to use an SQLite installation of our choosing, rather than one it would find by its own mechanisms. If we begin to distribute our own version of other underlying modules, the patches for the `_sysconfigdata` module would need to be modified to add support for those libraries.